Using AWK with OS-9

by

Zack C Sessions

ColorSystems
P. O. Box 540
Castle Hayne, NC 28429
(919) 675-1706
Quality OS-9 Software for the Color Computer 3
and the MM/1 from IMS

Using AWK with OS-9

by Zack C Sessions

ColorSystems

Chapter 1

Overview

In the February, 1992 issue of *The 68xxx Machines* [1], in his column Bob van der Poel talks about solving a complex problem with a few standard OS-9 commands along with a tool which does not come with OS-9 but is readily available. I was pleasantly surprised to see that Bob was talking about gawk. In my last semester of college I was required to get more "intimate" with awk for a course in software engineering. During that course I discovered a version of awk for the OS-9/68000 operating system and immediately saw its potential as a powerful tool for OS-9. Thus the justification to prepare this document.

First a little background. The awk programming language first came into being in 1977, being the brainchild of Alfred V. Aho, , and Peter J. Weinberger Brian W. Kernighan, all of Bell Labs. Knowing this, it is now obvious how the authors thought of awk as the name of the language. [2] The authors saw the power, yet shortcomings of the UNIX utilities grep, fgrep and sed. They adopted a goal of developing a pattern-scanning language that would understand fields, one with patterns to match fields and actions to manipulate them.

The current version of awk was released in 1985, and is documented in a book by Aho, Kernighan and Weinberger called *The AWK Programming Language* [3]. This version as implemented on most UNIX systems is

---

1. *Chatham House Company, Jim DeStafeno, Editor*
2. *Hint: Look at their last names, only!*
3. *Addison-Wesley Publishing, ISBN 0-201-07981-X*

called **nawk** for *new awk*, and the original 1977 version of awk remains as just **awk**. Nawk is a superset of awk.

The Free Software Foundation, as expected, released their version of awk called GNU AWK, or GAWK in 1986. As of this printing, the most recent version is V2.11. It supports all of the features and functions of awk as described in the aforemented book.[4] For your convenience a disk is included with the publication which contains the OS-9/68000 version of GNU AWK V2.11. Please read the license agreement in the file Copying in the GNU directory on the disk. Installation instructions are in Appendix A.

This publication uses example awk programs quite frequently. Again for your convenience all awk programs depicted in this publication are on the disk in the AWK directory.

If you've never heard of awk, you're probably asking yourself about now, *Just what the heck is awk and what can it do?* From here on out, I will refer to awk as awk but all programming examples will use the command required for OS-9, gawk, I will also be using the dollar sign (**$**) to indicate the Shell prompt in the example commands.

Awk is a very intelligent file processor. It can process several files with a single invocation. A typical awk command would look like:

```
$ gawk 'awk program' [file(s)]
```

If no files are listed, awk will read its input from the Standard Input path, making it perfect for receiving its data from a pipe, as you will see later. If more than one

---

4. *I have found a few situations where gawk does not fully function as described in the book, but nothing major.*

file is listed, all files are processed, but only one at a time, from left to right.

If the awk program is a multi line program, or you plan to use it often, you would want to create the awk program in a separate file and invoke the gawk command:

```
$ gawk -f awk.program [file(s)]
```

Where awk.program is the filename of the awk program file.

The key part of either format is the *awk program*. That is where all of the intelligence is at. An awk program is one or more awk program statements. Each statement has two parts, a pattern match, and an action. Both parts are optional, but one has to be there. So, the structure of a awk program statement would be:

```
[pattern] [{action}]
```

The square brackets indicate each part is optional, and actually, if omitted there is a default value for each of them. The action portion of an awk program statement is distinguished by being surrounded by curly braces. Each awk program statement in an awk program is applied to each record in the standard input file stream.

What this means is that the pattern is applied to the input record. If the pattern evaluates as true then the awk statement's action portion is executed. As you might expect, the syntax of a pattern contains only pattern matching types of commands and the action portion contains what might be referred to as executable code.

If an awk program statement has no pattern then the default pattern is applied to all records in the input stream. The default pattern is:

```
/*/
```

This *regular expression* will match all input records. This means that if a pattern is not specified then the action is applied to all records in the input stream.

If an awk program statement has no action then the default action is performed if the pattern ever evaluates true. The default action is:

```
{ print }
```

This action says to print the entire input record exactly as it was read in.

Here is a simple awk program:

```
{ print }
```

This awk program displays each record read to the Standard Output path. Since awk sends its output to the Standard Output path, it is perfect for piping into a child process. Here, there was no pattern, thus the default pattern *all records* was used.

Awk does some *pre-processing* of the records read from the input stream before passing the data to the awk program, in fact this is one of awk's most impressive and useful features. Each record is parsed and the *fields* are identified. Each field in the input is a string of characters which does not contain a blank. So the following line of data:

```
Kathy 4.00 10
```

Has three fields. The first field is a string and contains the value "Kathy". The second field is a numeric field and contains the value 4.00. The third field is also a numeric field and contains the value 10.

If an input line contains the string **"ABC DEF"** *(note the three spaces between the ABC and the DEF.)* then the first field would contain "ABC" and the second field would contain the value "DEF". The three spaces are essentially ignored by the field variables. In fact all characters which classify as *white space* are treated the same way by default.[5]

How does the awk programmer utilize the values of these fields in an awk program? Within the definition of the AWK Programming Language is a specification for a set of *builtin variables*.

One group of these builtin variables are used to reference the values of the fields in the *current input record*. The format for a field variable is **$n** where **n** is the field number. In the above record, **$1** would contain the character string "Kathy", **$2** would contain the number 4.00, and **$3** would contain the number 10.

Awk determines if the field is alphabetic or numeric. If a field contains a valid numeric value then it is a numeric field. Anything else is a string field.

There is another builtin variable name you can use to reference the entire input line. Variable **$0** refers to the original input record, including all white space characters which may have been ignored in the assignment of the field values. In this record, variable **$0** would contain the character string, **"Kathy 4.00 10"**. For the example above the variable **$0** would contain the string **"ABC DEF"**.

The **n** portion of the field variable can be a variable itself. For example, the following awk program will print each field of each record in a separate line:

---

5. *Using some of awk's special builtin variables, we can make use of certain white space characters for a useful purpose, as we'll see later.*

```
{ for (i = 1; i <= NF; ++i)
      print $i
}
```

Note again that the field's variable names start with 1. **$1** is the first field, **$0** is not. **$0** is the **entire** current input record. I'll talk about what **NF** is a little later.

You might also notice the similarity of awk action statements to C programming statements. That was no accident. It should also make it obvious that experiance with C programming will give one an edge at learning how to program awk actions. But that experience can also lead to potential problems. I have seen at least one implementation of awk which does not allow the following instruction in an action:

```
{
  for (i = 1; i <= NF; ++i)
     print $i
}
```

Notice the subtle difference? The first line contains only the first opening brace of the awk program. This is common coding style for most C programmers but as I have found out, can lead to problems with awk. It took quite a while to identify this as the reason the awk program was not working!

There are also a few other special builtin variables. **NR** contains the number of the current input record. Since an **END** pattern's action is executed after all input data has been processed, in an **END** pattern's action the builtin variable **NR** contains the number of records read from the input.

NF is the number of fields in the current record. So, the variable **$NF** would represent the **last** field in the current input record. There are several other builtin variables, all of which are beyond the scope of this article. The awk programmer can also use user defined variables

simply by referencing them, as I used the variable **i** in the action example above. The data type, string or numeric, is determined by the context. Arrays of numbers and strings are also supported.

All user defined variables also are automatically given an initial value the very first time they are referenced in an awk program statement. The initial value for numeric variables is 0, and for string variables "". Datatype is interpeted from the context it is used in.

The *scope* of variables is also important to realize. Simply put all variables are *global*, that is they are known in all patterns and all actions in the entire awk program, including possible **BEGIN** and **END** patterns. There is one anomoly. Parameters to *user defined functions* which I'll discuss later are passed by value for integer and string variables. Thus, while in the context of a user defined function the function is operating on a copy of the parameter and cannot affect the value of the global variable. Arrays are not copied, though, and can be modified in user defined functions.

The pattern portion is a little more difficult to fully understand, especially the regular expressions, unless you are familiar with the concepts of regular expressions a la UNIX. The pattern may be any one of the following:

1. The string **BEGIN**.
2. The string **END**.
3. An expression.
4. A regular expression.
5. A compound pattern.
6. A pattern range.

In this overview I'll touch on the first three. In a later chapter I'll go over the more complex patterns.

BEGIN and **END** are special patterns. They indicate that their associated actions (it is of little use to have a **BEGIN** or **END** with no action!), are only performed at

special times. The **BEGIN** action is **always** processed **before any** data has been read. The **END** action is **always** processed **after all** data has been processed.

A pattern which is in the format of an expression is a comparison between two expressions. All standard comparision operators used in the C Programming language are recognized, **<>, <=>, ==, !=, =, ,** plus two others, ~ means *is matched by*, and **!~** means *is not matched by*. I'll talk about these last two later. A typical expression for a pattern would be:

```
$2 <= $3
```

The pattern would be true if the value of field variable **$2** is less than or equal to the value of field **$3**. For each record in the input stream that the pattern is true, the pattern's associated action is executed. Since the default action is **{ print }**, if the above pattern were the entire awk program, then for each record which the pattern were true, the entire record would be written to standard output. Since expressions are allowed in the two items to compare, the following is a valid expression pattern:

```
$1 / 2 <= $4 * $5
```

In this case, field 1 is divided by a constant 2. That value is compared to the product of the values of fields 4 and 5, and the pattern is true if it is less than or equal.

A pattern can also be what is called a *string matching pattern*. In most cases, this is usually in the form is a single regular expression. To signify that a pattern is a regular expression, it must be enclosed in slashes. Here's an example of a simple regular expression:

```
/Mary/
```

In this case, the pattern is true for any record which contains the substring "Mary", and thus the pattern's

associated action would be performed. This is the simplest form of a regular expression. They can get quite complicated. A thorough treatment of regular expressions is beyond the scope of this publication. Any book on the UNIX Programming Environment will discuss regular expressions in depth. Even though, I will offer a few more examples of regular expressions in the chapter on patterns which do get a little more complicated.

The other part of an awk program statement is the action. An action consists of one or more valid awk action statements. These look much like C programming statements, and indeed, some are identical and function the same. Perhaps the most common action of most awk action clauses is to output something with either a print or a printf function. printf works exactly like the C function does. This print command:

```
print $1,$2,$3
```

would work like it were the following printf statement:

```
printf("%s %s %s\n",$1,$2,$3)
```

With the simple print statement the fields are automatically separated by a space. This example assumes that the three fields are all strings. awk is smart enough to know the difference. Note that the print also does an implied new line at the end of its data. Other action statements awk actions can have are if (with optional else), while, for, do while, break, continue, next and exit statements.

We'll revisit actions in a later chapter.

Chapter 2

Patterns and Regular Expressions

So far, I've touched on awk's command line syntax, what an awk program is and what it's basic structure is. We learned that each awk program statement has two parts, a pattern and an action. We also discussed a couple of different types of patterns. We talked about the special patterns **BEGIN** and **END**, about expressions as patterns, and simple regular expressions.

In this chapter, we'll first talk about some of the more complex pattern types and later we'll get into a more detailed discussion of regular expressions. Since the topic of this chapter is patterns, I will not be specifying any actions in any of the examples. Just keep it in the back of your head that if an awk statement consists of just the pattern and has no action, it still has an action, the default action. One more time, that action is:

```
{ print }
```

The first of the more complex patterns I will discuss are known as Compound Patterns. These are expressions which combine other expressions with logical ANDs, ORs, and NOTs. For example, you can have:

```
$1 == "Mary" && $3 > 100
```

Again, standard C operators are used, **&&**, **||** and **!** for and, or and not respectively. In this case, the pattern is true for any record where the first field contains exactly the string value "Mary" and the third field is greater than 100, when considered as a numeric variable. This reminds me of something I glossed over earlier. Let me digress on that for a second.

While a field which contains a *pure* numeric value is considered as a numeric field, it can be referenced in the context of a string variable. When used so, the value of the variable is converted to a string variable before the value of the variable is referenced. This is also true with alphanumeric variables, in fact. That is, a string variable refernenced in a numeric context, it's ASCII values are converted to an numeric value. The effect is analogous to an atoi() or an atof() function call in C.

Let me finish this digression by commenting that it should be obvious that variables are referenced more efficiently in the context of which the are defined as. This can make quite a difference when processing large data files as awk is an interpeted language.

Range Patterns are two patterns separated by commas. The range pattern is true for all records starting with a record for which the first pattern is true and then continuing sequentially through the file up and including a record, if found, for which the second pattern is true. If the first pattern is never true, no records will be processed. If the first pattern is ever true, and the second pattern is then never true, all records starting with the one which matched the first pattern through to the end of the input are processed by the range pattern's associated action. Each of the two patterns may be of any of the different types of patterns. For example:

```
/Alfred/, /Karen/
```

This is two regular expressions as the first and second pattern. In this case, the first record found which contains the substring "Alfred" and all records after it up to and including a record, if found, which contains the substring "Karen" are processed by the patterns' associated action. Another example of a valid range pattern is:

```
NR > 3, NR > 10
```

This shows that the two a parts of a range pattern can

be expressions as well as regular expressions. [6]

In this case the 4th through the 10th record are all processed by the patterns' action. I had to think about this one for a second. Remember, after the first pattern is true all records are processed UNTIL the second pattern is TRUE AND the input record which made the second pattern true is processed. Here's something similar:

```
$3 > 100, $3 < 200
```

In this case, if a record is found which has its third field greater than 100 then that record and all subsequent records will be processed by the action, until a record is come upon which has the third field less than 200. That record will be processed, too, but none after it. This one deserves a second thought also, but I'll let you handle that. Another interesting use for a range pattern would be this:

```
$1 == "Mary", $NF == 0
```

In this case, the first record found to have the first field equal to the character string "Mary" and all records after that will be processed until a record is found whose LAST field contains a value of zero. That will be also processed, but none after. Note that using the **NF** variable to denote the **last** field in a record, a file with variable numbers of fields in the records would be no problem.

A range pattern may not be part of another pattern.

Last time I talked about simple regular expressions. Let's get back into them. The type of pattern which contains a regular expression is called a *string matching pattern*. A pattern can contain more than one regular expression. It will contain either a single regular

---

6. *Remember the difference between an expression and a regular expression?*

expression, or a regular expression used in conjunction with an expression. To indicate a regular expression, it is surrounded by slashes. This string matching pattern must fit one of the following three general formats:

```
/regexp/
```

Matches with the current input line if the pattern matching rules applicable to the **regexp** are met.

```
expression ~ /regexp/
```

Matches if the string value of expression meets the pattern matching rules applicable to the **regexp**.

```
expression !~ /regexp/
```

Matches if the string value of the expression does not meet the pattern matching rules applicable to the **regexp**.

Any expression may be used in place of **/regexp/** in the context of ~ and !~. Note that the first format can be expressed as a valid form of the second format, that is:

```
$0 ~ /regexp/
```

Here's some examples of string matching patterns.

```
$4 ~ /Mary/
```

This is an example of type 2 above. In this case, field #4 of the input record must contain the substring "Mary" for the pattern to be true. Consider the following:

```
$1 ~ $3
```

In this case the **/regexp/** in the second format is replaced by an expression, in this case, the field variable $3. This can only be done in the context of ~ and !~. The ! operator is use as a **not** modifier. Here is an example of

its use:

```
$5 !~ /Phil/
```

This pattern would be true for all records which did **not** contain the substring "Phil". But, we have really only touched on what came come between the slashes for a regular expression. So far, all examples with regular expressions contained only a character string. There are many special characters called *metacharacters* which can be used to indicate special processing.

For example, the "^" character matches the beginning of a string and the "$" character matches the end of the string. These metacharacters may appear alone or in combination in a pattern. Consider the string matching pattern:

```
$1 ~ /^Chicago$/
```

In this case, the regular expression says to match with a string which starts with the C, and ends with the o, and has an hicag in between. So, only records whose first field is the string "Chicago" (not just contains the substring) will match and the pattern be true. The "*" character matches any size string of any characters, and the "?" character matches zero or one occurances of the previous character. Example:

```
$2 ~ /^Z*/
```

This pattern would be true for all records in which the second field **begins** with the character "Z" (UPPERCASE Z), followed by zero or more of any character. Consider this example:

```
$4 ~ /ing$/
```

This pattern would be true for all records whose fourth field **ends** with the substring "ing". Here's another:

```
$5 ~ /A?/
```

This pattern would be true for any record whose fifth field contained either the value "A" or "AA". The last metacharacter I will discuss is the [] pair. The [] contains one or more individually considered characters in it. It can also specify a range. For the pattern to be true, the string must match only the characters listed within the []'s. For example:

```
$1 ~ /^[ABC]/
```

This pattern is true for all records whose first field starts with one of the characters, "A", "B", or "C", and is followed by zero or more of any characters. Note that the comparison is **very** case sensitive! Here's an example of a range:

```
$2 ~ /^[a-zA-Z]/
```

This pattern is true for all records which has a second field which has as its first character a letter, upper or lower case. It may have zero or more characters after the initial letter. Be careful when using combinations of metacharacters! Consider the following example:

```
$2 ~ /^[a-zA-Z]*/
```

Now, at first glance, you might think that this does the same thing as the previous pattern. Uh, uh, it doesn't!! In fact, it will match on field two no matter what field two contains! You see, the first metacharacter is the dual character range which matches only a single character. Let's say that the range does match the first character. Then, no matter what is next, the "*" metacharacter will match it. But, let's say that the range does not match. Then no matter what is next the "*" metacharacter will match it! So, this pattern matches anything, which nullifies the reason to even attempt the first character verification. So, use the "*" metacharacter carefully!!

Multiple ranges may also be specified. For example:

```
$2 ~ /^[0-9][A-Z]$/
```

In this case, only records in which the second field starts with a numeric character and ends with an upper case alphabetic character.

Chapter 3

Actions

Awk would be a very dull language if all it could do is output the entire record of every record which is selected by a pattern! The action part of an awk statement is a very powerful tool, with it you can have the equivalent of an entire C program associated with each pattern in an awk program. There are 18 different awk action statements. Here is a brief description of each of them:

1. **`break`**

   This statement does the same thing as it does in a C program, it exits the current **`for()`** or **`while()`** loop.

2. **`continue`**

   This statement also does the same thing as it does in a C program, it transfers to the end of the loop and performs any end of loop processing.

3. **`delete`** *array-expression*

   This is something unique to awk. It is used to delete an entry in an array.

4. **`do`** *statement* **`while`** (*expression*)

   This is another awk action statment which is identical to it's C counterpart. The *statement* is performed as long as the *expression* is true, that is, it evaluates to a non-zero or a non-null value.

5. **`exit`** [*expression*]

   This statement functions as the C programming

language equivalent does, passing the value of the expression back to the caller. The specifics of the **exit** statement in an awk program is that:

1. If it occurs in an **END** pattern's action, the awk program is terminated and the expression is returned to the Shell.

2. If it occurs in any other action in an awk program it has the same effect as signalling an end of the input file stream. If there is an **END** pattern, its action will be performed.

4. *expression*

This allows for various arithmetic and string operations.

5. **if** (*expression*) *statement1* [**else** *statement2*]

This awk action statement also performs just as its C programming language counterpart. If the *expression* is evaluated as true then the first *statement1* is performed. Otherwise the *statement2* associated with the **else** *(if present)* is performed.

6. *input-output expression*

These statements allow for various input-output operations. Here are the various awk IO statements:

1. **close**(*expression*)

Closes a pipe or file denoted by *expression*.

2. **getline**

This is an all purpose statement to get input from a file, a variable or from the standard input stream.

3. **print**

This is one of awk's output statements. It allows the awk programmer to make simple awk programs when fancy output editting is not required.

4. **printf**

This statement works just as the C programming language counterpart does.

5. **system**(*command line*)

This statement also works just like its C counterpart.

7. **for** (*expression1*; *expression2*; *expression3*) *statement*

This is another awk action statement which performs the same as its C programming language counterpart. *expression1* is performed once, then the *statement* is performed repetatively as long as *expression2* evaluates as true. It is re-evaluated each time through before performing the *statement*. *expression3* is performed after the *statement* is performed each time *statement* is performed.

8. **for** (*variable* **in** *array*) *statement*

Now this one has no C programming language counterpart. It is quite unique to awk action statements. The *variable* is assigned each successive index value in the *array* and the *statement* is performed. The value of the items in the *array* can be accessed in the *statement* with the syntax *array*[*variable*].

9. **next**

Here is another awk action statement unique to awk. It is similar to the **exit** statement in that it affects the outer loop of the awk program which is reading the

input and parsing the awk program patterns and actions. Any time a **next** statement is performed, the next input record is read and parsed and control is passed back to the first awk program statement in the awk program.

10. **return** [*expression*]

    This awk action statement functions identical its C programming language counterpart. The exception is that in an awk program, the **only** place a **return** statement can appear is in a user defined function. I'll get into user defined functions later.

11. **while** (*expression*) *statement*

    This is another awk action statement which functions identical to its C programming language counterpart. The *statement* is performed as long as the *expression* is evaluated as true.

12. { *statements* }

    This structure is also permitted in the C programming language. It gives the programmer to utilize several awk action statements where only one is supported.

That is all of the awk action statements. At this time I want to go over the builtin functions since they are primarily used in actions. They are not restricted to just actions, though, they can be used in patterns as we will see later. There are two basic types of functions, string and arithmetic. In the following description of builtin functions *s* and *t* represent strings, *r* is a regular expression and *i* and *n* are integers.

String Functions

1. **gsub**(*r*,*s*,*t*)

   This function performs a global substitute of *s* for each

substring found in *t* which are matched by *r*, character for character with case sensitivity. It returns the number of substitutions which were made. The entire record, field **$0**, is used if *t* is omitted.

NOTE

If the character **&** appears in the replacement string *s* then it is replaced by the matched string, **\&** yields a literal ampersand.

2. **index**(*s*,*t*)

   This function returns the position in *s* where *t* appears. If it doesn't appear, a zero is returned.

3. **length**(*s*)

   This function computes and returns the length in characters of the string *s*.

4. **match**(*s*,*r*)

   This is the same function as **index()** except the starting point in *r and the number of characters in s* which are matched are determined by the builtin variables, **RSTART** and **RLENGTH** respectively.

5. **split**(*s*,*a*,*fs*)

   This is a special function which is unique to awk. It is used to split up the index values for an element in an associative array. I'll go over associative arrays in a later chapter. Formally what happens is that the array index value *s* is splitup into the string array *a* using the character *fs* as the field separator character. If *fs* is omitted then the builtin variable **FS** is used.

6. **sprintf**(*fmt*,*expr-list*)

This returns the *expr-list* formatted according to *fmt*.

7. **sub**(*r*,*s*,*t*)

   This function works just like the **gsub()** function described above except the substitution is performed only for the first match. Also, the note associated with **gsub()** also applies to this function.

8. **substr**(*s*,*i*,*n*)

   This returns a string which has the length of *n* which is a substring starting at position *i* in the string *s*. If the length *n* is omitted, the suffix of the string *s* starting at position *i* is returned.

Arithmetic Functions

1. **atan**(*y*,*x*)

   This function returns the arctangent in radians of *y* / *x* in the range of **- pi** to **pi**.

2. **cos**(*x*)

   This function returns the cosine of the angle *x* in radians.

3. **exp**(*x*)

   Returns the exponential function of e^x.

4. **int**(*x*)

   Returns the value of *x* truncated to an integer.

5. **log**(*x*)

   Returns the natural logarithm of the value *x*.

6. **rand()**

   Returns a pseudo-random number greater than 0 and less than 1.

7. **sin**(*x*)

   Returns the sine of an angle *x* expressed as radians.

8. **sqrt**(*x*)

   Returns the quare root of the value in *x*.

9. **srand**(*x*)

   Use this function to *seed* the random number generator. If no value for *x* is specified, the the time of day will be used.

That covers the *building blocks* of the awk programming language. Now how do we put them all together to produce useful awk programs? The answer lies ahead!

Chapter 4

Writing AWK Programs

In this chapter, I will concentrate on pure awk programming. I will use several example programs and also show how an awk program can be improved upon. In using awk programs as all of the examples, I will also be introducing other features and capabilities of awk which I have not previously touched on.

Since awk's real claim to fame is file processing, we'll need a few data files to play with. These are on the disk which is included with this publication in the AWK directory. To refresh your memory, here is the data contained in the two files we will be using in the examples.

```
$ list parts.dat
p1 nut red 12 london
p2 bolt green 17 paris
p3 screw blue 17 rome
p4 screw red 14 london
p5 cam blue 12 paris
p6 cog red 19 london
$ list emp.dat
Beth     4.00   0
Dan      3.75   0
Kathy    4.00   10
Mark     5.00   20
Mary     5.50   22
Susie    4.25   18
$
```

Program 1

Let's say we want to know how many times the character "o" appears in the file parts.dat. Consider the following awk program:

```
$ list P.4.1
{ for (i = 1; i <= length($0); ++i)
     if (substr($0,i,1) == "o")
       ++total
}
END { print "There were", total, "o's." }
```

If this awk program were used to process the parts.dat file, the output would look like:

```
$ gawk -f P.4.1 parts.dat
There were 9 o's.
```

This program introduces the builtin functions, of which both length() and substr() are members of. Both of these normally operate on string variables. Others operate only on numeric variables. In either case, if a variable is the wrong default type, its value is first converted to the appropriate value.

Program 2

Consider the standard word counting utility wc. When run against the parts.dat file, its output would be:

```
$ wc parts.dat
       6      30     131 parts.dat
```

Let's see how hard this is to do with awk. Take a look:

```
$ list P.4.2
{ nc += length($0) + 1
  nw += NF
}
END { printf("%7d%7d%7d %s\n",NR,nw,nc,FILENAME) }
```

There are two awk program lines in the awk program. The first program line does not have any pattern part, thus the action is applied to every record in the standard input stream. The second program line has a special pattern, the

keyword **END**, which means its action is performed just once, and only after all input records have been processed.

This produces output identical to the wc command above. Note how I use the **+=** additive operator, a la C. For those untrained in C the following two action statements are identical in effect:

```
nw += NF
nw = nw + NF
```

Also, remember that awk always initializes variables the first time they are referenced, thus the variables **nc** and **nw** are initialized to zero when the first record is processed. Remember also that **NF** is a builtin variable which contains the number of fields in the current input record.

Note also that the length of the entire record does not include the *newline* character at the end of every text record. That is why a 1 is added to the character count variable nc with each record processed.

Notice that the syntax of the printf() function is identical to the function of the same name in the C Programming Language. Also, a new builtin variable is being used, **FILENAME**. As you might expect, it contains the name of the file being processed.

Program 3

Consider the file parts.dat, it has one record for each part which is carried n each city of a multi location parts dealer. Let's say we want to know the number of parts in each of the cities. Take a look at the following awk program:

```
$ list P.4.3
   { tot[$5] += $4 }
END { for (city in tot)
        printf("City %s has %d total parts.\n",city,tot[city])
}
```

Here's what happens when we run it against the parts.dat file:

```
$ gawk -f P.4.3 parts.dat
City rome has 17 total parts.
City london has 45 total parts.
City paris has 29 total parts.
```

This program introduces an entirely new concept for awk programs. This concept is known as the "associative array", and is one of awk's most powerful tools. Here, the subscript for the array **tot[]** is the name of the city, a character string. Since there is no pattern, each input record is processed by the action. The number of parts in the city rome are accumulated in the array variable **tot["rome"]**.

Once all of the input records have been processed, we can dump out the totals in the **END** action. To access elements in an associative array, the awk programmer uses a special form of the **for()** loop, **for (item in array)**, where array is a previously defined associative array, and item will be a variable which will be assigned the value of each index in the array, one at a time. Thus, the variable item will be a character string variable. The value of each element in the array can be either numeric or alphabetic, in our example they are numeric.

Think of the C code it would be required to perform such a feat!

NOTE

Awk processes **all** arrays as associative arrays, that is, even if the index of an array is numeric,

the same awk internal processing code, the code which processes associative arrays. will process all references to members of that array.

Program 4

For our last awk program in this chapter, lets look at that parts file. Let's say that we want to know how many parts of each color are at each city. Consider this awk program:

```
$ list P.4.4
   { total[$5,$3] += $4 }
END { for (idx in total) {
        split(idx,x,SUBSEP)
        printf("There %s %d %s part%s in %s.\n",
            (total[idx] == 1 ? "is" : "are"),total[idx],x[2],
            (total[idx] == 1 ? "" : "s"),x[1])
    }
  }
```

Here's what happens when it's run:

```
$ gawk -f P.4.4 parts.dat
There are 17 green parts in paris.
There are 17 blue parts in rome.
There are 12 blue parts in paris.
There are 45 red parts in london.
```

This awk program again uses an associative array, and in this case, it is a two dimensional array. First dimension is the city name, and the second dimension is the part color. The value of the elements in the array is the total number of those parts.

Apparently, awk actually processes the multiple indexed array by simply concatenating the two indices together into a single character string where the two values are separated by a special separator character. One argument for this is that to reference the index values separately in a subsequent **for()** loop, the awk

programmer must split up the values by using a new builtin function, **split()**. Also used is another builtin variable **SUBSEP**. This variable contains the default character used to separate subscript values for an associative array.

Notice also, that awk supports a powerful C expression, the *conditional expression*.

Extra!

Lastly in this chapter, I'll present a "real world" example. Let's say I have a directory tree of many files in several directories. I realize that all of the files have public write access and I do not want that. I want to remove public access from the files. I could do the following:

```
$ dir -sur ! attr -z -npw
```

But this would change the access mask of all of the subdirectory files in the tree! I did not want to do that! Consider the following awk program:

```
$4 !~ /^d/ { print $7 }
```

OK, for any input record in which the fourth field starts with a lower case d, that record is ignored. If the fourth field starts with anything else, the the seventh field of the record is written to the standard output path. What good is that? Well consider the following OSK command:

```
$ dir -sure
  0.0     92/12/27 1200   d-ewrewr      2E        64 DIR1
  0.0     92/12/27 1150   ---w--wr       4     10476 file1
  0.0     92/12/27 1210   ---w--wr      3A      1004 DIR1/file2
```

The s option tells dir not to sort the output, this makes the command run faster and sorted output is not really needed for our ultimate purpose. The u option means the output is unformatted, basically this means no

header information, just one record of output per file. The r option tells dir to do a recursive search. You'll notice for files in the listing, the full path name is displayed for files below the current level. The e option asks for a full, or entire, directory listing.

Now, notice that field 4 is the attributes mask. For directory files, this mask will **always** have a "d" in the first position. Conversely, all files which are not directory files will have a dash ("-") in that position. Now, notice that the file name is the seventh field in the display. Now that awk program is starting to make sense! What if I entered the following command:

```
$ dir -sure ! gawk '$4 !~ /^d/ {print $7}' ! attr -z -npw
```

Just what the doctor ordered!

Notice also that the same effect could be produced with either of the following awk programs:

```
2. $4 ~ /^-/ {print $7}

3. substr($4,1,1) == "-" {print $7}

4. substr($4,1,1) != "d" {print $7}
```

Now, given the fact that the same function can be performed with several different awk programs, *which form would be the most efficient?* Well, it should be obvious that the first and second form which use the regular expression would be almost identical in efficiency, as the only difference is the direction of the comparison. The same holds true with the two forms which use the **SUBSTR()** builtin function.

To find out which of the two basic forms was the more efficient and to verify my assumptions above, I prepared a test file which contained a total of 2268 lines, 2225 were non-directory files and the remaining 43 were directory files. Here are the timing results:

1. 35 seconds
2. 35.5 seconds
3. 38 seconds
4. 38 seconds

As one can see, my assumption as to the closeness of the two groups which were similar held true, even more so for the **SUBSTR()** form. We see also that, even though the improvement is small, the form which uses the regular expression was almost 9% faster. Granted, in my example a difference of three seconds is not all that much, but let's say you had a file with 100,000 records in it! Your savings in processing time would be 2.5 minutes!

Chapter 5

Advanced Programming

In this chapter we'll get a little more advanced in our programming examples. We will do this while concentrating on text processing with awk. Awk is a pretty nice text file processor for certain needs.

For our first example this time let's revisit an awk program from the previous article on awk programming, the wc lookalike awk program. Our awk program could emulate wc just fine, but only for one file at a time. wc is capable of working on several files with the same command, and providing an overall total. For example look at the following command and its output:

```
$ wc parts.dat emp.dat
      6      30     131 parts.dat
      6      18     112 emp.dat
     12      48     243 total
```

Remember, the parts.dat and emp.dat file were defined in the previous chapter.

Now, what do we need to do to the awk program from last time to get it to be able to do this? As it is, when used with the same two files, it produces the following results:

```
$ gawk -f P.4.2 parts.dat emp.dat
     12      48     243 emp.dat
```

It gives the right overall total, but doesn't give the totals for the individual files, and the filename displayed for the total is the name of the second (really the last) file to be processed. Consider the following modification:

```
$ list P.5.1
name != FILENAME { disp() }
name == FILENAME { nc += length($0) + 1
                   nw += NF
                   nr = FNR
                 }
END { disp()
      if (numfiles > 1)
          printf("%7d%7d%7d total\n",gt_nr,gt_nw,gt_nc)
    }
function disp ( dummy ) {
    if (NR > 1) {
        gt_nr += nr
        gt_nw += nw
        gt_nc += nc
        printf("%7d%7d%7d %s\n",nr,nw,nc,name)
        nw = nc = 0
        ++numfiles
    }
    name = FILENAME
}
```

With this awk program, we are introduced to many new topics. First of all, this is the first awk program in my series which has more than one statement, other than a **BEGIN** or **END**. In this case, we have two awk statements, each with unique explicit patterns.

Notice that the **NR** builtin variable is not used in this version as it was in the previous version. This is because in this version, we want to handle multiple files. If multiple files are specified, the builtin variable NR takes on a more refined definition. It's formal definition is *input record number since beginning*. There is another builtin variable **FNR**, used here whose formal definition is *input record number in current file*. Also realize that the formal definition for **FILENAME** is *name of current input file*. The meaning of the term *current file* should be obvious in the context of processing a list of files.

This awk program also introduces the concept of user

defined functions. In this case a function used to display the totals for the previous file is used since the code for it is needed in two separate actions in the awk program, the second pattern statement, and the **END** statement. Note, that even though no argument is passed in either call to disp(), it is defined with a single argument which I called dummy.

According to *The AWK Programming Language*, a function with a null argument list is perfectly valid. But there appears to be a bug in the GNU version of AWK. It will correctly process a call to a function with a null argument list, but it does not appear to be able to declare a function with a null parameter list. Therefore, the function disp has a single argument declared which is never used.

Now, let's get down to something a little more complex, shall we? Let's say, for example, you have a text file which is the output of a text formatting utility. This document file could have lines that are specially formatted with centered, filled and adjusted lines such as:

```
These Lines are Lines
Which Have Been Centered
For Emphasis

Other lines would have a certain amount of
spaces at the beginning of the line to give
the text an even left margin. There are also
embedded spaces between some words so that
we can also get an even right hand margin
also, as this paragraph demonstrates.
```

Notice that the last line in a paragraph is not filled out to the right border. The first line of each paragraph may or may not be indented more than the other lines in the paragraph. There are also blank lines inserted in the text so that when printed on a printer, each page's text appears at the same location as all pages do.

Now, lets say that you do not have the original file which was the input to the text formatter, and you want to regain that file from the formatted file. Well, if it is a large file, several minutes in an editor will fix it for you. Or you can fix it in a few seconds with the editor and then with the help of the following awk program.

First thing to do is to prepare the file with an editor. Lines which are centered as the above lines only need to be left justified on column 1, like so:

```
These Lines are Lines
Which Have Been Centered
For Emphasis
```

To signal that lines are centered lines and should not be automatically filled, we edit the file an insert a line before them and a line after them, both lines having a single character, a percent sign (%). For example, the above lines would be:

```
%
            These Lines are Lines
          Which Have Been Centered
                For Emphasis
%
```

Next thing to do is to remove the extra lines which form page breaks. If they occur in the middle of a paragraph, be sure to remove them all. If the occur in between two paragraphs, then be sure to leave a blank line between them. Once this is done, the file is then ready to be processed by this awk program:


```
$ list P.5.2
{ if ($0 == "%") {
    if (flag == 0)
      flag = 1;
    else
      flag = 0;
  }
  else {
    if (flag == 1) {
      for (i = 1; i <= NF; ++i) {
         printf("%s",$i);
         if (i < NF)
           printf(" ");
      }
      printf("\n");
    }
    else {
      if (length($0) == 0) {
        if (length(rec) > 0) {
          printf("%s\n",rec);
          rec = "";
        }
        printf("\n");
      }
      for (i = 1; i <= NF; ++i) {
         if ((length(rec) + length($i)) > 50) {
           printf("%s\n",rec);
           rec = "";
         }
         if (length(rec) > 0)
           rec = rec " ";
         rec = rec $i;
      }
    }
  }
}

END { if (length(rec) > 0)
        print rec
    }
```

The preceeding example text after being processed by this AWK program would appear as:

```
These Lines are Lines
Which Have Been Centered
For Emphasis

Other lines would have a certain amount of spaces
at the beginning of the line to give the text an
even left margin. There are also embedded spaces
between some words so that we can also get an even
right hand margin also, as this paragraph
demonstrates.
```

Notice that all extraneous spaces between words have been eliminated, there being but a single space between words. Words have been moved up from lines to fill out lines to a maximum of 49 characters. Of course, you can modify the constant **50** to whatever value you are shooting for. To use this awk program one would enter the following command:

```
$ gawk -f P.5.2 input.file >output.file
```

Let's analyze the awk program. First thing we notice is that there is no **BEGIN** pattern but there is an **END** pattern. In fact most of the awk program consists of but a single action with no pattern match at all. This means that this action will be applied to each record in the input file. And the action itself resembles a small C program.

The first thing the program does is to see if we need to turn on or off the word fill flag variable which is named **flag**. So, if the input record is nothing but a **%** character, then the value of the flag is checked. The first time this is done (for the first input record), the value of flag is automatically initialized to zero. So, it is changed to 1. If the value is 1, it is changed back to zero. Also, if this record was a flag record, it is not processed further, that is, it is not written to the standard output.

If the input record is not a fill flag record, it is then parsed. The method of the parse depends on the state of the variable flag. If it is a 1, then no word fill is performed. Each field is written to the output, with a single space being output after each one except the last. After the last field is written out, a newline is written out. Now, one may ask, why not just write out field **$0**. That wouldn't work since field **$0** is the entire input line, which includes the extra spaces at the beginning of the line and the possible character(s) between some of the words. When awk parses an input line and sets the values of the field variables, preceeding and successive *white space* is ignored. It is this technique of parsing by awk which this program takes advantage of.

Let us now look at how lines are filled when the fill flag has the value of zero. First thing to do is see if we have a paragraph break. This is signalled by a line which has no characters. It actually has a single character, a newline character, but this is not counted in the length of the input record. If it is a paragraph break, we must then check the length of the variable **rec** which is being used to build output lines. If it is more than zero, we it is the last line of a paragraph and needs to be written out, along with a newline character. The variable is also set back to a null value. If there is anything or not in rec, we still need to write out a newline.

The next **for()** loop could have been done in an else clause of the preceeding **if()** but if the length of field **$0** is zero then **NF** is also zero, so the **for()** loop will not be executed anyway if field **$0** has a length of zero. If the input record does have any fields, ie, in this case words, then we process each of the fields one by one.

In the field processing loop, the first thing we need to do is check the length of the output variable, rec. If it's length would exceed 50 characters if the current field were concatenated to it, then the value is output to the standard output and its value is reset back to a null value. Note that the first time **rec** is ever referenced it is initialized to a

null value.

Whether or not the value of **rec** was written out, we now process the current field. If the length of **rec** at this time is more than zero, then it contains at least one word, but is too short for a full line. This means we need to concatenate a space on the end of it before be concatenate the current field to the end of it.

The **END** pattern is required to handle the situation of some value being in **rec** after all input record have been processed.

Chapter 6

Parting Thoughts

This is as deep as I want to get with regular expressions. I'll end this time with a quickie awk program which may help to illustrate a technique. You want to know how many total bytes are used by the files in the current directory. Consider the command:

```
$ dir -eu ! gawk '{ tot += $6 } END { print "Total bytes", tot }'
```

While the actual use of this command is a waste of time if you have a copy of the ls command which can supply file size totals, it illustrates how you can interpet system function displays by awk programs. In this case, the 6th field, the size in bytes, is summed to the variable tot. At the end of file, the total is displayed.

This example also shows an arithmetic expression I haven't mentioned, ie, the use of the "+=" arithmetic operator. As you might expect, all C type arithmetic expressions are supported in awk programs, including the auto pre or post increment.

Now, at the risk of making Bob mad at me, I am going to analyze his awk programming skills using his February, 1992 article as a guide. His first awk program is a simple one,

```
$1 ~ /bsr/ { print $2 }
```

What Bob is doing here is one step in a series of commands. In this step all he wants to do is to parse an assmebler source listing and find all of the subroutine references. If it is a subroutine call, then the first field of the record will contain "bsr" for *Branch to Subroutine*. The

second field will contain the name of the subroutine.

Well, nothing I can say about that. Short, sweet and functional. But looking ahead I see that this awk program is intended to be run on several different files by means of a cfp procedure and since it's short, it is being run supplied on the command line itself. In a UNIX environment, that is fine, but on the MM/1 using gawk, there is something to consider. Each time you run a command like:

```
$ gawk '/Asia/ { print $3,$4*$5 }' countries
```

there will be a file created in the /dd/TMP directory which contains the awk program. **Each** time is is run, a **separate** file is created, even if the awk program being executed over and over and over is exactly the same program. So, every so often, you need to clear out the /dd/TMP directory. So, in Bob's procedure, for this reason, I would have extracted the awk program for both gawk commands out into an external file and use the -f option.

Next program, I got a few observations. Bob uses a **BEGIN** pattern merely to initialize a variable to 0. Since all variables are given an initial value the very first time they are ever referenced, and if they are used in the context of a numerical expression, they assume the value 0. So, the **BEGIN** pattern and its action are redundant and not needed. I can't really improve on the rest of the second awk program.

I will finish up this chapter with an awk success story which strengthens Bob's premise in his article, that is to use the tools you have. I had just downloaded several files and I wanted to set the attributes of the files to public read/write. These files were the only files in the directory which were created on that date, but there were other files in the same directory I didn't want to mess with. So, consider the following command:


```
$ dir -e ! gawk '$2 == "92/03/11" { print $7 }' ! attr -z -pr -pw
```

Actually, that is not how I first did it. My first attempt was even cruder:

```
$ dir -e ! grep 92/03/11 ! gawk '{ print $7 }' ! attr -z -pr -pw
```

Actually, if grep is already in memory, the second version would probably run faster, um, nope, I was thinking we would save a lot by having grep do the pattern matching, but gawk will still parse the entire record before even processing the pattern, in this case, process all records. But, the overhead of creating the fourth process needs to be considered also. It was pretty to watch in a procs display in another window!

We have seen now the text processing power of awk, and even more complex operations can be performed using the **substr()** function. I hope that this publication on the awk programming language has helped you to get a good start with awk!

Appendix A

Installing GAWK

GNU AWK consists basically of just the executable binary file, *gawk*. It is located in the CMDS directory on the disk included with this publication. Be sure to read the copyright and license notice of the Free Software Foundation. It is the file *Copying* in the GNU directory. The program files used as examples in this publication and the data files to run them with are in the AWK directory. .You may copy them anywhere you wish. That directory also has a few extra awk programs in it not used as examples in this publication.

Full documentation and sources for gawk are available from the Free Software Foundation. See the aforementioned license file. But I would personally recommend getting a copy of the book I mentioned in Chapter One, since it is the definitive reference on the Awk Programming Language written by the authors of AWK itself.

Just copy the gawk program binary file to whichever CMDS directory you wish to put it in and be sure the execution permissions are set for your system, ie, you may or may not want to set the public execution attribute. You will also need to create a directory in the root of your /dd device called TMP. This also implies that you have to have a /dd device. For most OS-9 users, this is equivalent to their primary hard disk device, usually /h0. This directory is used to store "temporary" source files when you invoke the gawk command and the AWK program is part of the command line enclosed in single quotes, eg,

```
$ gawk '{print $1}' foo.bar
```

After a command like this finishes, you will find a new file in the /dd/tmp directory called gawkxxxxxx, where the xxxxxx is some unique number. Probably due to an

oversite of the programmer who ported GNU AWK to OS-9, gawk leaves these files after they are used, they are not deleted, so periodically you will need to manually delete them.

There is also an environment variable which you may choose to use. Whenever you run gawk using the -f command line option, eg,

```
$ gawk -f prog.awk foo.bar
```

gawk normally looks for the AWK program file, prog.awk in this example, in your current data directory. If you have the environment variable AWKPATH defined to a valid directory, that directory will be searched for the AWK program file. This environment variable can set with the setenv command, eg,

```
$ setenv AWKPATH /h1/usr/zack/awk
```

In a timesharing environment, this command is normally done in the user's *.login* file. Also, remember that environment variable names are case sensitive, and AWKPATH **must** be in UPPERCASE. A minor annoyance with this technique, however, is if the AWKPATH environment variable is set, gawk will want ALL awk programs you reference in that directory. If you want to run an AWK program which is located in some other directory, but still want to have the AWKPATH environment variable defined, you must force gawk to look elsewhere for the AWK program file. You can either specify a full path for the program file, eg,

```
$ gawk -f /dd/usr/zack/prog.awk foo.bar
```

or, you can specify a relative path, eg,

```
$ gawk -f ./prog.awk foo.bar
```

In this case, the current data directory is searched for the AWK program file.

Appendix B

AWK Limitations

Awk does have some limitations. The following table is the formal definition of the limitations of awk as described in *The AWK Programming Language*. It may or may not reflect the actual limitations of the implementation of awk you may be using. I have not verified that gawk matches these limitations.

100 fields
3000 characters per input record
3000 characters per output record
1024 characters per field
3000 characters per printf() field
400 characters maximum literal string
15 open files
1 pipe
double precision floating point